First-Class Undefined Classes for Pharo

Guillermo Polito, Stéphane Ducasse, Luc Fabresse

First-Class *Undefined Classes* for Pharo

From Alternative Designs to a Unified Practical Solution

Guillermo Polito
RMoD - Univ. Lille, CNRS, Centrale Lille, Inria, UMR 9189 - CRIStAL - Centre de Recherche en Informatique Signal et Automatique de Lille, F-59000 Lille, France
guillermo.polito@univ-lille1.fr

Stéphane Ducasse
Inria Lille-Nord Europe, 40 Avenue Halley, Villeneuve d'Ascq, France
stephane.ducasse@inria.fr

Luc Fabresse
IMT Lille Douai, Univ. Lille, Unité de Recherche Informatique Automatique, F- 59000 Lille France
luc.fabresse@imt-lille-douai.fr
http://car.imt-lille-douai.fr/luc

CCS Concepts **• Software and its engineering → General programming languages**; **• Theory of computation** → *Program analysis*

Abstract

Loading code inside a Pharo image is a daily concern for a Pharo developer. Nevertheless, several problems may arise at loading time that can prevent the code to load or even worse let the system in an inconsistent state.

In this paper, we focus on the problem of loading code that references a class that does not exist in the system. We discuss the different flavors of this problem, the limitations of the existing Undeclared mechanism and the heterogeneity of Pharo tools to solve it. Then, we propose an unified solution for Pharo that reifies Undefined Classes. Our model of Undefined Classes is the result of an objective selection among different alternatives. We then validate our solution through two cases studies: migrating old code and loading code with circular dependencies. This paper also presents the integration of this solution into Pharo regarding the needed Meta-Object Protocol for Undefined Classes and the required modifications of existing tools.

Keywords Dynamic Languages, Partial Code Loading, Reflection

1. Introduction

This paper explores the problem of loading incomplete, partial or broken code because of inexistent classes into a dynamic object-oriented language such as Pharo [DZH+17].

Loading code inside a Pharo image is a daily concern for a Pharo developer. While doing this daily task, there are several situations in which a loaded package may refer to classes that are not defined in the runtime. For example, let us consider a library that worked well in an older Pharo version, say Pharo 2.0. Porting this library to the latest Pharo 6.0 requires to load it in the newer targeted Pharo version. However, this will surely provoke several loading errors because the old library code references classes that existed in Pharo 2.0 but do not exist anymore 4 years later in Pharo 6.0. These classes could have been renamed, removed or modified. The same situation occurs when loading libraries from other Smalltalk dialects to port them. Another example occurs when we try to load a library without loading all its dependencies because we do not know them beforehand or because we do not know the correct loading order. This happens when the library developer did not document properly the dependencies or did not use a package management tool such as Metacello [BCDL13].

References to undefined classes exist in different flavors: a package can contain subclasses of undefined classes, methods extending undefined classes, or simply methods referencing undefined classes. Trying to load such *incomplete* code in Pharo produces many different load errors depending on the used tool. Some tools do fail, some partially load the package, some others silently modify the code to make it loadable. For example, while the Monticello version control system will reject the creation of classes whose superclass does not exist, a *file-in* ignores the inexistent superclass and makes the new class subclass of Object.

In this paper we propose to represent *Undefined Classes* as stub classes that are automatically installed in the system when the need for a non-defined class is detected. *Undefined Classes* enables the loading of incomplete, partial or broken code, and are helpful to keep track of potential bugs. Furthermore, when loading a class for which an *Undefined Class* already exists in the environment can be correctly handled without any information loss such as method extensions.

The paper is organized as follows. Section 2 digs further in code loading problems and gives an overview of the Pharo 6 current state regarding this problem. Section 3 presents three different models to represent *Undefined Classes*. It then describes an analysis to select the most suitable model for Pharo according to their integration within Pharo or their ability to load partial code that can be correctly handled afterwards without any loss of information. Section 4 evaluates our implementation of undefined classes by using them in three different scenarios: the migration of old code, the loading of code without its dependencies, and the loading of code with circular dependencies. Before concluding this paper in Section 7, Section 5 discusses the runtime and tools integration of *Undefined Classes* in Pharo.

2. Problems when Loading Code

2.1 Motivating Example

Let's consider a package Package that extends the package Dependency (cf. Figure 1). Dependency defines the Platform class that represents the current platform and an FFI class that can be used to call C functions. Package extends Dependency in two ways: it defines several concrete subclasses of Platform, and it extends Platform with the extension method workingDirectory. The following source code illustrates this example:

```
Platform >> workingDirectory (extension)
    ^ self subclassResponsibility

Platform subclass: #Windows
    instanceVariableNames: ''.

Windows >> workingDirectory
    ^ FFI call: '_getcwd' library: LibC

Platform subclass: #Unix
    instanceVariableNames: ''.

Unix >> workingDirectory
    ^ FFI call: 'getcwd' library: LibC
```

Let's imagine now that we want to load Package without loading Dependency. If we do so, three problems may arise depending on the used tool. First, the superclass of Windows and Unix is not defined and the classes cannot be properly created. Second, the extension method Platform>>workingDirectory cannot be loaded because the class it belongs to does not exist. Third, classes Windows and Unix have methods that use the FFI class, not existing either. Generally speaking, we identify three kinds of broken class references:

Undefined superclasses. A class inherits from a non-defined class.

Undefined class references. A method has a reference to a non-defined class.

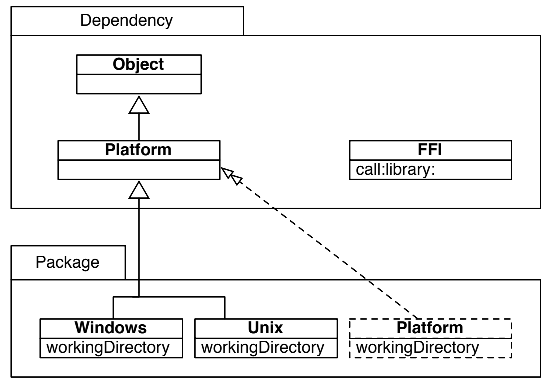

Figure 1. Package extends Dependency by subclassing, extension methods and uses it through direct references.

Undefined extended classes. An extension method is meant to extend a non-defined class.

2.2 Current State in Pharo

We have identified three different kinds of broken code. However, each tool in charge of loading code inside Pharo produces different results in the presence of such cases. In this section we compare how the Monticello version control system, the Code Importer tool and the Pharo IDE (i.e., the Nautilus browser) manage each of the cases we described before. Table 1 summarizes our analysis.

Tool	Superclasses	References	Extensions
Monticello	Ignored	Undeclared	Ignored
Code Importer	Override	Undeclared	Error
Nautilus	Forbidden	Forbidden	Forbidden

Table 1. Comparison of strategies to manage broken code by Pharo tools

Monticello. Monticello is the open-source Smalltalk version control system used in most Pharo versions up to today. Monticello throws a so called warning when loading a subclass or an extension method of an undefined class. If the user manually creates the missing classes at this point and resumes the process, Monticello still ignores the newly created classes and does not load the problematic subclasses/methods. To overcome this issue, the user needs to manually create the missing classes and restart the loading process from scratch. A different approach is taken with class references in methods. In those cases, the reference will be loaded as an *Undeclared* reference in the Pharo global environment pointing to nil and will be fixed as soon as the class is loaded and the method recompiled.

Code Importer. The code importer is the tool in charge of loading code from files (also named file-ins). When we do a file-in of a subclass whose superclass does not exist, the code importer will silently override the superclass. That is, it loads the class as a subclass of the default superclass of the system (e.g., Object or ProtoObject). A completely different behaviour is exposed when the file-in includes extension methods to undefined classes: the code importer will raise a MissingClassException. Finally, when loading methods referencing to undefined classes, the code importer behaves as Monticello and it defines an *Undeclared* reference.

Nautilus. Nautilus is Pharo's current code browser and main IDE tool used to manipulate code. Nautilus avoids by design the creation of classes or methods referencing to undefined classes. When trying to create a subclass or reference to an undefined class, Nautilus interactively suggests to use an existing class or to create a new class or a global variable. Regarding extension methods, a method can only be created directly on the target class. Thus, if the class does not exist, the method cannot be created on the first place and we cannot define it as an extension method.

It is worth noting that, with the exception of the created *Undeclared* references, all other behaviors are silent and leave no track of potential problems. This hinders the task of identifying and thus fixing this problems once the code is loaded (or not).

2.3 Problem Statement

Code that extends or references undefined classes should be loadable and potential problems should be identifiable to postpone their resolution. Our aim is to simplify the tasks of browsing and modifying code when we ignore the dependencies of a package or when migrating a package to a newer version of the platform. An ideal solution should manage all three kinds of broken references: subclasses, extension methods and method references. But it should also ensure the correctness of loading a class for which previous unresolved references have been loaded. In other words, an ideal solution should support loading code containing broken references and then correctly fix them afterwards if the actual class is finally loaded. Moreover, the solution should be unified, compatible and used by all existing tools.

3. Solution: First-Class Undefined Classes

We propose the introduction of undefined classes as first-class entities in the language runtime. First-class undefined classes solve all the problems stated before. It solves the problem of code loading: we can create subclasses of undefined classes, we can use them as placeholders for extension methods, and undeclared references can point to them. It also provides explicit tracking information: the system can be queried for all undefined classes to detect potential bugs and problems. Finally, a first-class entity provides an entry point for existing tools.

In the rest of this section we present alternative designs for undefined classes. Finally, we analyse and compare them to choose the most suitable model to introduce into Pharo.

3.1 A Unique Undefined Class

At first sight, the simplest design to introduce undefined classes is to create a single UndefinedClass class (cf. Figure 2). Classes that need to extend an undefined class either by subclassing or method extension will subclass from this class and store the corresponding extension methods into it. *Undeclared* references can also point to this instance instead of nil, providing a much better control in the case of broken code being executed.

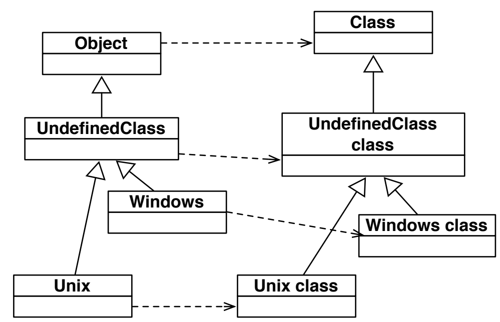

Figure 2. Design 1: Introducing a single UndefinedClass class. Classes inheriting from an undefined subclass will be loaded as subclasses of this class.

The major drawback of this design is that it is a partial solution since UndefinedClass is a shared object. Indeed, the name of the original unknown class is lost both in the superclass or extension methods cases. Of course, we considered some extensions of this design to store those runtime informations elsewhere such as in inside separate data structures. However, we excluded such extensions from our analysis because they do not comply with the object-oriented paradigm and will be costly to maintain in the long term.

3.2 Undefined Class's Instances

To solve the information loss problem, another solution based on the previous solution is to introduce one UndefinedClass instance per undefined class. UndefinedClass instances store all information related to a single undefined class *e.g.,* the name of the original subclass, its subclasses, loaded extension methods. *Undeclared* references will now reference to instances of these classes, providing an even more fine-grained control on code execution.

Two different designs emerge from this idea:

1. modeling only the class instance-side
2. modeling both instance and class-side as a pair of instances

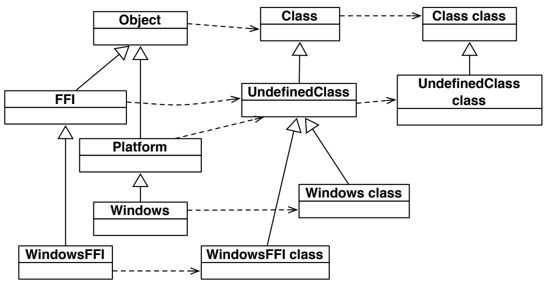

Figure 3. Design 2: One instance of UndefinedClass per undefined class.

For presentation purposes, we illustrate in Figure 3 only the case of modeling the instance-side of classes. Figure 3 shows also one of the main problems of this solution: its complexity. By creating instances of UndefinedClass, the meta-level is shifted: instances of UndefinedClass should be subclasses of Object and thus UndefinedClass needs to be a (indirect) subclass of Class like other classes. The second alternative design, introducing first-class meta-classes, proposes a place to store class-side extension methods while it shifts the meta-level even another time.

In essence, both of these designs are complex and make programs harder to understand to developers. It is also a major drawback for tools that should manipulate yet another kind of entity (UndefinedClass instances) non polymorphic with classes such as Traits. This drawback suggests that we should carefully design UndefinedClass API to be polymorphic with classes.

3.3 Undefined Classes Subclasses

A third possible design is to create one UndefinedClass subclass per undefined subclass. Creating a normal class creates a class and implicitly its metaclass pair. Similarly to the previous designs, UndefinedClass subclasses fulfill both the role of a stub for *Undeclared* references, and also present a natural place to store all information related to the non-defined class, including the class name and its extension methods.

Figure 4 depicts the main difference between this solution and the previous ones that resides in the final structure of the class hierarchy. Indeed, this solution does not shift the class-hierarchy, making it easier to understand and analyse. Moreover, in this solution undefined classes are just regular classes created using the conventional class-creation mechanisms. This simplifies significantly the interaction of undefined classes with existing tools, since by design they are classes and behave as such.

The main drawback of this solution is that any UndefinedClass will behave as a normal defined class and produce silent misbehaviours. Indeed: can an UndefinedClass be instantiated? Does it make sense to put an UndefinedClass in a package? Of course, the regular class API can be redefined

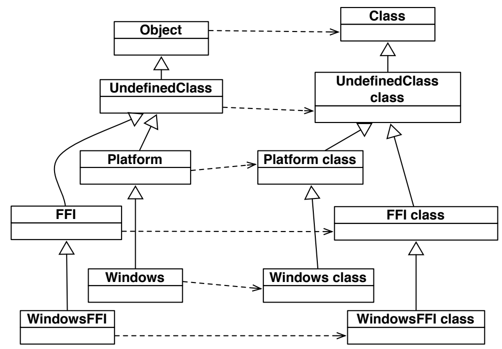

Figure 4. Design 3: One subclass of UndefinedClass per undefined class.

in UndefinedClass. However, preventing inherited behavior from Class in UndefinedClass subclasses is not good for substitutability and therefore tools uniformity. This design also implies that all future modifications of regular classes should be considered from the UndefinedClass perspective and correctly treated.

3.4 Selecting the most suitable model for Pharo

To select the best model, we evaluate the described designs using the following criteria:

Subclasses Support. The solution supports the loading of subclasses (✔) of undefined classes or not (✗).

Extensions method Support. The solution supports seamlessly loading extension methods (✔), loading extension methods is possible but requires implementing special support for it (~) or it is not supported (✗).

Class references Support. The solution supports to load and differentiate *Unreferenced* class references (✔), it supports to load but not differentiate *Unreferenced* class references (~), or it does not support loading *Unreferenced* class references (✗).

Retained Information. The solution seamlessly retains a majority of runtime information such as class names, subclasses, extension methods (✔), retaining runtime information is possible but it requires implementing special support for it (~) or all information is lost (✗).

Base/Meta-levels consistency. As a descendent of Smalltalk, the Pharo model is organized along a parallel hierarchy of classes. Each class as its own metaclass [GR83, BDN$^+$09]. This uniform structure is important to ensure not introducing inconsistencies between the base and meta-levels. The solution is compliant with this model (✔) or not (✗).

Tools compatibility. This criteria makes reference to how tools support each of the designs e.g., if undefined classes are navigable with Nautilus, mergeable/committeable/loadable with Monticello, writable and readable with CodeImporter. A solution may seamlessly interact with existing tools (✔), it would require partial adaptations (~) or the support for it should be entirely implemented (✗).

Criteria	Design 1	Design 2	Design 3
Subclasses	✔	✔	✔
Extension Methods	✗	~	✔
Class References	~	✔	✔
Information loss	✗	~	✔
Levels Consistency	✔	✗	✔
Tool Compatibility	✔	✗	✔

Table 2. Comparison of strategies to manage broken code by Pharo tools

Table 2 presents a summary of the evaluation of these criteria for our three designs.

Design #1. This design fails to represent undefined classes as single entities, provoking the loss of all runtime information related to the class including extension methods.

Design #2. This design improves on **Design #1** by adding first-class undefined classes. However, with this solution the class hierarchy becomes significantly more complex. This hinders not only understanding but also requires the adaptation of the undefined classes meta-model so we can integrate it with existing tools.

Design #3. This design improves on **Design #1** by not losing runtime information and on **Design #2** by simplifying the class hierarchy to the level of the normal class hierarchy. That makes it a design that can easily and seamlessly integrated with existing tools.

Finally, we selected the **Design #3** to model undefined classes in our solution.

3.5 Ensuring Loading Correctness

Representing unresolved references as first-class entities is not enough to ensure loading correctness. Our full solution provides additional support. When loading code, the first time an unknown class name is encountered, our solution creates a new UndefinedClass subclass to represent this missing class. Then, when this same name is again encountered, the same UndefinedClass subclass is reused and completed with additional information such as new method extensions. When loading a class for which a subclass of UndefinedClass exists in the system, our solution first creates the class and then correctly introduce all previously loaded definition stored in its UndefinedClass subclass placeholder before destroying it.

In essence, the loading correctness of our solution relies on: using only one UndefinedClass subclass per missing class (identity), UndefinedClass subclass automatic migration, and destruction when loading the actual class and system uniformity i.e., all code loading mechanism relies on our system.

4. *Undefined Classes* in Action

In this section we explore the usage of our chosen design in two different scenarios. Our first scenario shows how we can load and migrate old code to a newer code base. Our second scenario shows how we can load two circular-dependent packages separately without losing any code.

4.1 Scenario 1: Migrating Old Code

In this section we show how we used undefined classes. In this scenario, we tried to several old packages that are not supported on the latest version of Pharo (6.0) anymore.

Algernon. Algenon is an old Pharo package used to navigate existing methods, classes and packages. In the latest Pharo6 release, we loaded Algernon using its Metacello configuration from the project catalog. The last commit of this configuration is from 24 March 2015 (ConfigurationOfAlgernon-FN.8). We found in this package a class TypeList that is subclass of RectangleMorph. RectangleMorph has beed removed in Pharo 3.0. To load such package we executed the following expression:

```
ConfigurationOfAlgernon project load: #bleedingEdge.
```

Seaside 2.8. Seaside is a Smalltalk web framework that makes emphasis on the creation of components and has the novelty of using continuations. We loaded Seaside 2.8 using the metacello configuration in the project catalog. This version dates from 10 October 2011(ConfigurationOfSeaside28-dkh.40). We found that this version depends on the current version of Pharo by creating several subclasses of PackageInfo and several extension methods defined in the classes BlockContext and ContextPart. BlockContext was removed before Pharo 2.0, PackageInfo was removed in Pharo 4.0 and ContextPart was renamed to Context in Pharo 4.0. To load such package we executed the following expression:

```
ConfigurationOfSeaside28 load.
```

Omni Browser. Omni Browser is the code browser used in Pharo before Nautilus was introduced in 2012. We loaded Omni Browser using the metacello configuration in the project catalog. This version dates from 20 August 2015 (ConfigurationOfOmniBrowser-pad.187) and was probably maintained for the Squeak dialect and not Pharo. We found that this version defines a subclass of TextMorphEditor and several extension methods defined in the classes ClassOrganizer, MethodReference and BorderedSubpaneDividerMorph. TextMorphEditor, MethodReference and BorderedSubpaneDividerMorph were removed before Pharo 2.0, and ClassOrganizer was removed in

Pharo 3.0. To load such a package we executed the following expression

```
ConfigurationOfOmniBrowser project latestVersion load
```

This particular scenario included also initialization code in the form of class side initialize methods. These class side initialize methods are automatically executed when a package is loaded. So, when they contain a reference to an undefined class, a runtime error will happen while executing such method. In such case, the developer must manually fix the code that caused the error in the open debugger and resume the code loading without losing any runtime information. An unexperienced developer that does not know the loaded project, could simply comment the problematic piece of code to analyse it later on. For further work, we will analyze wether this is a good default behavior, or if packages containing undefined classes should not automatically execute their class side initialize methods.

4.2 Scenario 2: Loading Order

The order used to load packages is important to correctly resolve references. Using our undefined classes model this is not a constraint anymore. It is also useful to packages with circular dependencies.

You can see this in the following code:

```
1  Seaside3LoadingTest>>
       testLoadingSeasidePackagesInRandomOrder
2    | maxNumberOfUndefinedClassesCreated |
3
4    self assert: UndefinedClass allSubclasses isEmpty.
5    maxNumberOfUndefinedClassesCreated := self
       loadSeasidePackagesInRandomOrder.
6    self assert: maxNumberOfUndefinedClassesCreated > 0.
7    self assert: UndefinedClass allSubclasses isEmpty.
8
9    self executeInitializeClassMethodsInCorrectOrder.
10
11   self assertAllSeasideUnitTestsAreGreen.
```

This code is an experiment of loading all the packages of Seaside 3 and its dependents (59 Monticello packages in total for the stable version) but in a random order (line 5). It means that a lot of undefined classes are created and correctly resolved later on because at the end of the loading step, there is no remaining undefined class (line 7). During the loading of each package, we disabled the initialization of classes. On line 10, we currently trigger all class side initialize methods in the right order as defined by the project's configuration. We are currently working on doing this automatically when a class is complete. Finally, on line 11, we execute all Seaside unit tests (more than 800 unit tests) and ensure that they pass.

5. Pharo Integration

This section first presents the runtime integration through the required Meta-Object Protocol for Undefined Classes, and then the tools integration.

5.1 Runtime: *Undefined Classes* MOP

Our implementation of *Undefined Classes* in Pharo is publicly available under the MIT License. Our UndefinedClass implementation is rather small and easy to understand: it mainly consists in one class with 3 methods and 14 green unit tests. The following code snippet shows how to load it in the latest Pharo 6.0 using:

```
Gofer new
  smalltalkhubUser: 'StephaneDucasse' project: 'PetitsBazars';
  package: 'ClassParser';
  package: 'UndefinedClasses';
  load.
```

Our design makes it easier to load code that is potentially broken. Indeed, automatically creating class stubs to represent a class opens the door to instantiate such a class, or to create a subclass with an unexpected format. This opens several questions: should we allow the creation of instances of undefined classes? In case we do, how should these instances respond to messages? Our main criteria to answer this question was to avoid silent solutions: an undefined class is indeed broken code and the developer should be notified of the mistakes he makes to be able to solve them.

Instantiation. We decided that undefined instances should not be instantiated because they represent a partial class definition that probably miss some initialization code. To enforce this, we redefined the method basicNew in UndefinedClass class-side such that it throws an error.

```
UndefinedClass class >> basicNew
   ^ UndefinedClassError signal:
      'Cannot instantiate undefined class: ', self name
```

Thus, any try to instantiate the UndefinedClass or a subclass will fail at runtime.

Messages. Our solution needs to handle messages to UndefinedClass instances even if we forbid by design their creation. Indeed, developers may create such new instances using other mechanisms such as using the change class primitive. For example, the following piece of code creates a normal instance of Object and then changes the class of such object by SomeUndefinedClass using the adoptInstance: message.

```
object := Object new.
SomeUndefinedClass adoptInstance: object.

object class => SomeUndefinedClass
```

To cover to some extent such behaviour, we redefined UndefinedClass>>doesNotUnderstand: to throw a notifying error. If an (sub-)instance of an UndefinedClass is created in the system, messages to it will then fail accordingly.

```
UndefinedClass >> doesNotUnderstand: aMessage
  UndefinedClassError signal
```

Notice that defining doesNotUnderstand: does not cover the creation of instances. basicNew is defined on the class-side, while doesNotUnderstand: is defined on the instance-side. Moreover, we did not use this same doesNotUnderstand: mechanism on the class side because UndefinedClass inherits from Class. doesNotUnderstand: cannot simply trap basicNew since it *understands* all messages of a class. We could have used a more sofisticated class proxy such as in Ghost [PBF+15] at the expense of a more complex implementation and having some impact on the compatibility with tools (Section 5.2).

5.2 Tools

To provide a coherent behaviour across the entire runtime, development tools should be updated to use this mechanism. As part of this work, we identify the adaptation points in the tools, required to support undefined classes. Our modifications of Pharo core classes and tools can be loaded with:

```
"Patch the whole Pharo 6.0 system to use our implementation"
Gofer new
  smalltalkhubUser: 'StephaneDucasse' project: 'PetitsBazars';
  package: 'UndefinedClassSYSTEMPATCH';
  load.
```

We present here the adaptations of three main Pharo development tools:

Monticello. Monticello requires two main modifications to support the loading of undefined classes. First, MCClassDefinition and MCMethodDefinition, the objects part of Monticello's meta-model, should be extended to create the corresponding undefined classes at load time. Second, the class MCPackageLoader makes a pre-load analysis to split a package between loadable and unloadable definitions. Because of the introduction of undefined classes, such separation is not valid anymore and Monticello needs to be modified accordingly.

CodeImporter. Code importer requires also two main modifications. First we needed to adapt how a chunk of code containing an expression (*i.e.,* a *do-it chunk*) is interpreted. A *do-it chunk* is an region of a file in the file-in file format that contains an expression. *Do-it chunks* contain arbitrary expressions and also class definitions that will be just evaluated by the compiler. In the case of an undefined class, at code compilation time the name of the undefined superclass is replaced by a reference to nil, making it impossible for tools to recover the class name in an efficient and simple manner after the compilation. To solve this issue we introduced a class parser at the level of the code importer before any code evaluation. The class parser allows us to distinguish if a given expression is a class definition or not, and so create undefined superclasses before creating the subclasses. On the other hand, extension methods in Code Importer are managed as in Monticello: we only require a patch to create the expected extended class.

Nautilus. We decided to leave Nautilus behaviour as it is. We believe Nautilus behaviour, though conservative and limited, is explicit, clear and easy to understand for developers.

6. Related Work

Multiple IDEs and tools provide nowadays the possibility of loading broken code and undefined classes. This is the case, for example, of Smalltalk's refactoring browser parser [RBJO96], Eclipse for Java [Ecl03] or the XText platform to create programming languages [Bet13]. The design of these tools is fault-tolerant. Thanks to this they support robust syntax highlighting, code analyses and code navigation. The main difference between their approach and ours is that they work exclusively on a static representation of the code. Indeed, a program with errors cannot be compiled, executed nor tested until all it's errors are fixed beforehand. Our undefined classes model is instead a runtime model: we allow loading classes at runtime and execute code over them.

Callau et al. [CT13] propose also a similar approach to support test driven development (TDD) [Bec02] on mainstream IDEs. They reify in the IDE undefined entities and use them as stubs instead of the real classes. These reifications allow developers to work on the design of their APIs without fighting against constant compilation errors. Moreover, once the design phase is finished, developers can use these stubs to automatically generate the classes corresponding to their design. This approach does indeed reify undefined classes as in our approach but with a different objective in mind. However, they apply their approach to augment mainstream IDEs, having also the limitations of working on a purely static environment.

7. Conclusion

This paper addresses a daily concern of Pharo developers: loading code that contains unresolved class references. Current Pharo version (6.0) does not correctly handle this is common problem that arises when loading old or cross-dialects libraries. In this paper, we have proposed a model for first-class *Undefined Classes* to represent missing classes in the system. An *Undefined Class* stores all required informations that should not be lost while the real class is not loaded such as its superclass name or extension methods added by other packages. Afterwards, when the missing class definition is finally loaded, it can be completed with all informations previously stored in the *Undefined Class* that were representing it. Our design of *Undefined Class* is polymorphic with classes making it easier to integrate in Pharo and tools that manipulate classes. Of course, an *Undefined Class* is not a regular class since this is a partial definition and that is why we added it has its own specific MOP.

This work will continue along two directions. First, we plan to finish the integration of *Undefined Classes* into Pharo 7.0. Then, this effort is part of a larger one aiming at providing a module system for Pharo. Modules must define their dependencies. When loading a module, class references inside a module may not be resolved until we bound them.

References

[BCDL13] Alexandre Bergel, Damien Cassou, Stéphane Ducasse, and Jannik Laval. *Deep Into Pharo*. Square Bracket Associates, 2013.

[BDN+09] Andrew P. Black, Stéphane Ducasse, Oscar Nierstrasz, Damien Pollet, Damien Cassou, and Marcus Denker. *Pharo by Example*. Square Bracket Associates, Kehrsatz, Switzerland, 2009.

[Bec02] Kent Beck. *Test Driven Development: By Example*. Addison-Wesley Longman, 2002.

[Bet13] Lorenzo Bettini. *Implementing Domain-Specific Languages with Xtext and Xtend*. Packt Publishing, 2013.

[CT13] Oscar Callau and Eric Tanter. Programming with ghosts. *IEEE Softw.*, 30(1):74–80, January 2013.

[DZH+17] S. Ducasse, D. Zagidulin, N. Hess, D. Cloupis Originally written by A. Black, S. Ducasse, O. Nierstrasz, D. Pollet with D. Cassou, and M. Denker. *Pharo by Example 5*. Square Bracket Associates, 2017.

[Ecl03] Eclipse platform: Technical overview, 2003.

[GR83] Adele Goldberg and David Robson. *Smalltalk 80: the Language and its Implementation*. Addison Wesley, Reading, Mass., May 1983.

[MPBD+11] Mariano Martinez Peck, Noury Bouraqadi, Marcus Denker, Stéphane Ducasse, and Luc Fabresse. Efficient proxies in Smalltalk. In *Proceedings of ESUG International Workshop on Smalltalk Technologies (IWST'11)*, Edinburgh, Scotland, 2011.

[PBF+15] Mariano Martinez Peck, Noury Bouraqadi, Luc Fabresse, Marcus Denker, and Camille Teruel. Ghost: A uniform and general-purpose proxy implementation. *Journal of Object Technology*, 98:339–359, 2015.

[RBJO96] Don Roberts, John Brant, Ralph E. Johnson, and Bill Opdyke. An automated refactoring tool. In *Proceedings of ICAST '96, Chicago, IL*, April 1996.